VŠB TECHNICKÁ UNIVERZITA OSTRAVA
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Vizualizace Ray Tracingu

Visualization of Ray Tracing

Tadeáš Popov

Bakalářská práce

Vedoucí práce: Ing. Tomáš Fabián, Ph.D.

Ostrava, 2021

Abstrakt

Tato práce se bude zabývat vizualizací cest paprsků ve scéně s možnostmi filtrování paprsků dle zadaných parametrů a zobrazení informací o konkrétním dopadu paprsku. V dnešní době začíná být ray tracing populárnější a s častějším používáním přichází i více chyb, které je třeba odstranit. Tyto chyby však není lehké odhalit, a proto je nutný program, který by dokázal pomoci chyby identifikovat. Na trhu je k dispozici několik technologií k debugování, ale většina neumožňuje sledovat konkrétní cestu paprsku. Rádi bychom v práci dosáhli možnosti výběru části obrazovky, které umožní vykreslit paprsky letící do těchto pixelů. To opět umožní ještě lepší možnost pro debugování, protože pokud je na obrázku vidět, kde se odehrává něco špatně, stačí pouze provést výběr obrazovky a cesta paprsků se zobrazí. K řešení bude využito JavaScriptu a knihovny třetí strany ThreeJS, také bude využit jazyk C++ pro vytvoření loggeru cest paprsků.

Klíčová slova

6 bakalářská práce; ray tracing; ThreeJS; C++; logování; ladění

Abstract

This bachelors thesis will deal with the visualisation of the ray paths in scene with the options to filter the rays with parameters and to show each ray hit info. Nowadays, ray tracing has begun to be very popular and used, but mistakes are being brought with that. Therefore, some program is needed which could help find these mistakes. Even though there are some programs available for debugging they mostly can not do the path ray tracing. The goal of this thesis is to achieve an option, which would make it possible to select some rectangle on the screen and it would show paths of rays flying into those pixels. This would make debugging easier if the user knows where the mistake is located in the picture he/she can just select the rectangle which contains it and he/she would immediately see the ray tracing path. JavaScript and third-party library ThreeJS will be used for the solution of this problem also, there will be used the programming language C++ for the creation of ray paths logger.

Keywords

6 bachelors work; ray tracing; ThreeJS; Vue; C++; logging; debugging

Poděkování

Rád bych poděkoval svému vedoucímu práce Ing. Tomáši Fabiánovi, Ph.D., a všem, kteří mi s prací pomohli, protože bez nich by tato práce nevznikla.

Obsah

Seznam použitých zkratek a symbolů

RT	–	Ray Tracing
RTV	–	Ray Tracing Visualiser
JS	–	JavaScript
HTML	–	Hyper Text Markup Language
API	–	Application Programming Interface

Seznam obrázků

Kapitola 1

Úvod

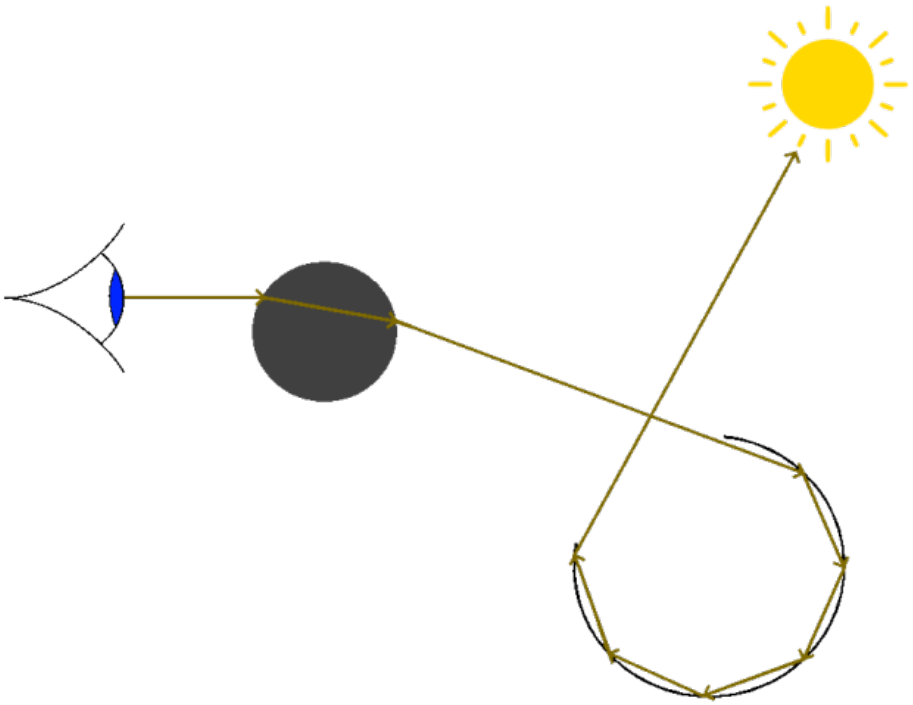
Tato práce se zabývá vizualizací ray tracingu, bude sloužit jako návrh a bude implementovat řešení pro jednodušší hledání chyb v ray tracerech. Důvodem zvolení si tohoto tématu byl můj zájem o ray tracing, který zdánlivě působí na první pohled velmi komplexně, ale nakonec se ukázalo, že ray tracing není až natolik složitý. Obsahem této kapitoly bude představení ray tracingu, ray traceru a problematiky hledání chyb. V následujících kapitolách budou představena existující řešení, jejich srovnání a následně bude prezentováno vyhodnocení srovnání. Práce bude pokračovat návrhem projektu a vysvětlením cíle projektu. Následovat bude seznam použitých technologií, zaznamenání běhu ray traceru a popis implementace uživatelského prostředí. Bude zde také vysvětlen princip vyhledávání paprsků. V závěru bude shrnuto měření rychlosti vyhledávání paprsků, a zároveň budou uvedeny návrhy pro vylepšení. Nakonec bude zrekapitulován a demonstrován celkový výsledek této práce.

Ray tracing [1] (dále jen RT) je metoda sledování cest paprsků ve scéně. Touto metodou dosahujeme reálnějších snímků ve hrách a filmech. V reálném světě se paprsky světla odrážejí od předmětů až nakonec doletí do našeho *oka* [2], díky čemuž vidíme. RT funguje podobně, pouze v opačném směru. Paprsky se sledují z našeho oka na každý pixel na obrazovce. V principu se jedná o jednu z technik řešení *Detekce skrytých ploch* [3], což RT řeší tím, že hledá nejbližší povrch pro pohledový paprsek. A to je ekvivalentní setřízení všech objektů per pixel.

Ray tracer je tedy program, který pro každý pixel počítá odrazy daného paprsku. Takové výpočty ve velkém množství mohou trvat poměrně dlouho, proto ray tracery běžně využívají více vláken procesoru, každému vláknu je přitom možné přiřadit sekci (obvykle obdélníkového tvaru), pro kterou bude vlákno paprsky počítat. Tímto způsobem lze celkový běh ray traceru podstatně urychlit. Při obyčejném sledování cesty paprsku se může stát, že se ray tracer zacyklí na jednom paprsku, neboť odrazy se mohou teoreticky počítat donekonečna. Z toho důvodu ray tracer musí nějakým způsobem implementovat kontrolu hloubky paprsku. Jedním možným řešením je po každém odrazu snižovat průchodnost vrcholu a v momentě, kdy úroveň průchodnosti vrcholu dosáhne přednastavené hodnoty, ukončit výpočet pro daný paprsek. Dalším možným a implementačně jednodušším

způsobem je nastavit tento limit absolutně. Problém přichází ve chvíli, kdy se tento limit nastaví například na deset zásahů a ray tracer pro konkrétní paprsek vygeneruje například sto zásahů. Úkolem této práce je nabídnout řešení, které zásáhá a uloží informace o paprscích a jejich zásazích z ray traceru. Toto řešení zároveň umožní takové paprsky vyhledat a vykreslit ve scéně.

Běžné monitory mají Full HD (1920×1080) rozlišení a ty lepší mohou mít dokonce až 4K (3840×2160), což znamená, že pro běžný monitor musí ray tracer spočítat minimálně 2 073 600 paprsků a pro lepší monitor až 8 294 400 paprsků. S tímto přichází velký problém debugování, neboť scéna je složitá, nachází se v ní mnoho objektů a takto je chyba pro člověka sotva odhalitelná v jednom paprsku, natož ve dvou milionech. Paprsků je velké množství a navíc jeden paprsek se může skládat z vícero zásahů. Každý zásah paprsku nese informaci o svém počátku (odkud se odrazil), směru (kam letí) a také vzdálenost od počátku. Výpis takových informací (navíc v tak velkém množství) do terminálu (pokud by byl buffer v terminálu schopen udržet takové množství dat) uživatele zahrne příliš mnoha daty a snadno se v takovém výpise ztratí.



Obrázek 1.1: Cesta paprsku ve scéně. Na tomto obrázku je zobrazeno, jak může vypadat cesta konkrétního paprsku z oka do světla. Paprsek nejprve prochází průhlednou koulí, ze které se následně odrazí do zrcadlového oblouku, přes který se dostane až ke zdroji světla

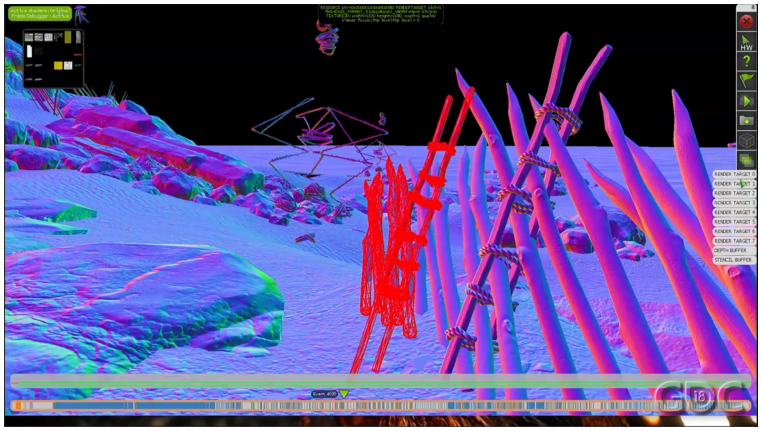
Kapitola 2

Přehled existujících řešení

V předchozí kapitole bylo nastíněno, co je to ray tracing a ray tracer samotný. V této kapitole budou představena již existující řešení, která se taktéž snaží řešit problém debugování ray tracerů. Zároveň zde budou popsány jejich výhody, nevýhody a důvod, proč je naše řešení lepší.

2.1 NVIDIA Nsight Graphics

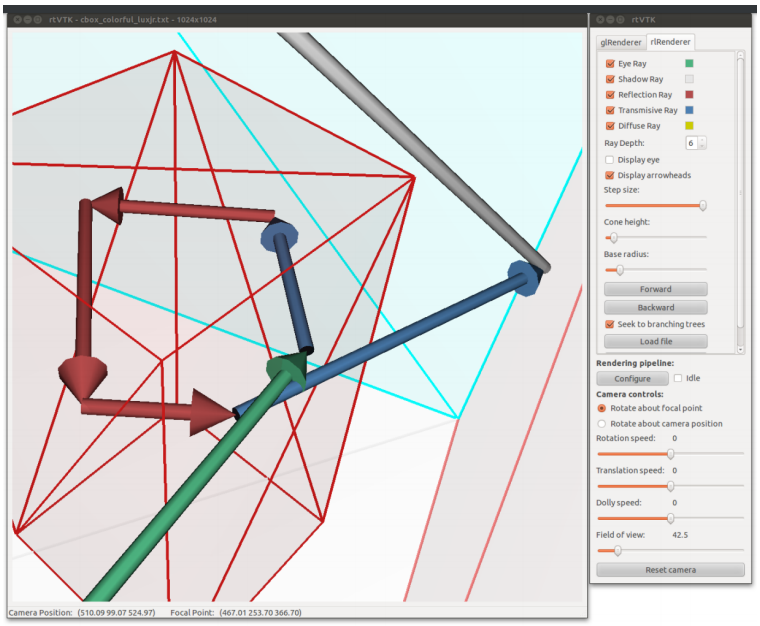
Jedním z hlavních představitelů debugování je nástroj *NVIDIA Nsight Graphics* [4]. Toto řešení je velice rozsáhlé a nezabývá se pouze ray tracingem. Obsahuje nástroje pro návrhy zlepšení výkonu, debugování spadnutí grafické karty, dokonce zobrazuje i informace o prostupu jednotlivých jednotek grafické karty a disponuje funkcemi pro celkové debugování jednotlivých snímků. Co se týče RT, je schopno barevně rozlišit množství dopadu paprsků. Hlavní nevýhodou oproti našemu řešení je, že zde není možnost sledovat cestu konkrétního paprsku.



Obrázek 2.1: Uživatelské rozhraní NVIDIA Nsight Graphics. Na pravé straně lze opět vidět panel nástrojů. Nad spodním okrajem je zobrazena lišta s posloupností vykreslování. Scéna je vykreslena tak, že červená barva značí nejvíce osvětlené plochy a zelená nejméně

2.2 rtVTK

Mezi další konkurenci lze zařadit *rtVTK* [5]. Tento projekt nabízí přesně to co potřebujeme, až na několik drobností. Za běhu ray traceru jsou této knihovně předávány informace o paprscích a knihovna je poté schopna tyto paprsky zobrazit na obrazovce. Největší nevýhodou této knihovny je, že poslední aktualizace pro knihovnu vyšla v roce 2014, takže je zdánlivě zastaralá. Mezi další podstatné nevýhody patří nemožnost vyhledávat v paprscích a závislost na dalších knihovnách (CMake v2.8, Qt v4.8.5 SDK, OpenGL Utility Toolkit v3.7, OpenGL Extension Wrangler v1.8...). Výhodou i nevýhodou této knihovny je přímé použití v ray traceru.



Obrázek 2.2: Uživatelské rozhraní rtVTK. Na obrázku vpravo lze vidět panel nástrojů s možností filtrace konkrétních zásahů paprsků dle jejich typu a hloubky a nastavení jejich vzhledu. Pod tímto se nachází nastavení pro kameru. Levou část obrázku zabírá samotná scéna

2.3 Analýza problému

Výsledkem tedy je, že na trhu aktuálně není připravený žádný debugovací nástroj, který by nebyl příliš složitý na použití a nebo nebyl příliš zastaralý, díky čemuž má tato práce smysl a využití, neboť tento projekt nabízí jak jednoduchost, tak i aktuálnost. Další funkcionality, které by celou práci lehce pozvedly o úroveň výš, mohou být vždy přidány v budoucnu.

Kapitola 3

Návrh projektu

V předchozích kapitolách byly uvedeny problémy vyskytující se při ray tracingu a také byla představena některá již existující řešení těchto problémů. V této kapitole bude nadefinován návrh projektu, který bude potřeba pro řešení zmíněných problémů a celkový cíl bakalářské práce.

3.1 Cíl projektu

Cílem projektu je navrhnout a implementovat vizualizační nástroj umožňující jednodušší debugování ray traceru pomocí zobrazení cest jednotlivých paprsků a možností jejich vyhledávání. Tím pádem bude nejprve potřeba nějak zaznamenat běh ray traceru a jeho výstupu. Pro tento účel bude sloužit Logger, který bude zapisovat všechny zásahy paprsků do jednoho nebo více souborů. Jakmile budou k dispozici data, se kterými můžeme pracovat, budeme potřebovat nějaké uživatelské rozhraní, které programátorovi nabídne možnosti vyhledávání v paprscích a jejich vizualizaci. Programátor by měl mít také možnost se pohybovat ve scéně a kontrolovat jednotlivé zásahy. Pro tuto část využijeme možnosti webového prohlížeče, který nám umožňuje vykreslit jak samotnou scénu s paprsky, tak i prvky pro ovládání a interakci s uživatelským rozhraním.

3.2 Zápis paprsků

Pro vývoj Loggeru bude použit programovací jazyk C++, neboť právě v tomto jazyce jsou ray tracery psány. Tím zajístíme kompatibilitu Loggeru s ray tracerem. Dále bude Logger podporovat více vláknový zápis, protože tato funkcionalita je často využívána ray tracery pro rychlejší vykreslování a výpočet.

Jak již bylo v úvodu zmíněno, paprsků je mnoho a mohou obsahovat různé informace. Pro jejich uložení se nejvíce hodí formát JSON, ve kterém budou paprsky vypočítané ray tracerem uloženy. Tento formát se navíc skvěle hodí pro nahrávání do webového prohlížeče, kde se používá

jazyk JavaScript, který již v základu disponuje možností tento formát přeložit do JavaScriptového objektu či pole.

3.3 Uživatelské prostředí

Takový soubor nebo soubory budou vybrány a nahrány uživatelem ve webovém prohlížeči a budou využity k zobrazení informací o konkrétních zásazích a také v nich budou vyhledávány požadované paprsky. Při zobrazení mnoha paprsků by scéna mohla začít vypadat velice chaoticky, z tohoto důvodu by měl být zaveden limit pro počet vykreslených paprsků. Aby se uživatel v paprscích lépe vyznal, bylo by dobré jednotlivé zásahy paprsků zobrazit jinou barvou. Konkrétní zásah paprsku by měl být představován pomocí koule. Po kliknutí na tuto kouli by se uživateli mělo zobrazit okno s detaily o zásahu paprsku. V případě, že se v jednom místě sejde více zásahů, měl by mít uživatel možnost konkrétní kouli reprezentující zásah schovat a opět zobrazit.

Kapitola 4

Použité technologie

V následujících částech budou popsány použité technologie, které budou potřeba k implementaci návrhu z předchozí kapitoly. Taktéž zde budou uvedeny důvody, z jakých byly konkrétní technologie zvoleny.

Oproti ostatním existujícím řešením byla pro naše řešení zvolena cesta webového prohlížeče. Tím je získána výhoda v jednoduchosti použití, ale zároveň také nevýhoda omezené paměti, kterou může prohlížeč využívat.

C++ je rychlý, kompilovaný *objektově orientovaný* [6] jazyk [7], který je hojně rozšířen. Jak již bylo zmíněno, C++ byl zvolen, neboť ray tracery jsou také obvykle psány v C++, a proto bude Logger v C++ více vyhovující kvůli jednoduššímu použití v ray traceru. I přesto, že se jazyk C++ dá považovat za složitější a celkově vývoj v něm není tak rychlý, Logger nebude obsahovat natolik složité funkce, aby se v tomto jazyce nestihl jednoduše napsat.

Intel Embree je knihovna napsána v jazyce C++ a je především mířena pro programátory pracující na ray tracingu využívající procesory Intel [8]. Pro nás je důležité, že obsahuje připravené struktury paprsků pro ray tracery. V Loggeru jsou struktury inspirovány touto knihovnou a jsou využity k předávání informací o paprscích a následnému zápisu do souboru/ů.

JavaScript (dále pouze JS) je interpretovaný, *prototype-based* [9] jazyk [10] kompilovaný při spuštění s *first-class* [11] funkcemi. JS je také jedno-vláknový a *event-driven* [12] - což ulehčuje práci s uživatelskou interakcí. JS je používaný (nejen) ve webových prohlížečích, ale je používán i například v *Node.js* [13]. Jelikož se uživatelské rozhraní nachází ve webovém prohlížeči, JS je tedy nejvhodnějším jazykem.

nlohmann JSON je C++ knihovna [14] umožňující použít formát JSON v C++, který v základu neobsahuje funkce pro práci s tímto formátem. Logger bude tedy využívat tuto knihovnu pro uklá-

dání informací o paprscích a následném zápisu do souboru/ů. Existují i další formáty, ve kterých lze paprsky zapsat, například XML nebo YAML. *YAML* [15] obsahuje dobře čitelnou syntaxi a je možno v něm ukládat různé datové typy včetně JSONu. Pro naše řešení není ale čitelnost důležitá a JS v prohlížeči nativně neobsahuje funkce pro převod z formátu YAML do JS pole. XML [16] má sice oproti formátu YAML podporu v JS, ale ve srovnání s JSONem je redundantní, což je problém, protože paprsků, jak již bylo zmíněno, je mnoho a to by mohlo vést k větším souborům, než pomocí JSON formátu.

Vue je JS framework [17], ulehčující práci s uživatelskou interakcí, ale také pomáhá rozdělit komplexní webovou stránku do jednotlivých komponent. Umožňuje rozdělit JS kód na *direktivy* [18], *komponenty* [19], *mixiny* [20] a *pluginy* [21]. Tímto způsobem se kód stane podstatně přehlednějším a také umožňuje znovu použít již naprogramované věci rychle a jednoduše. Tento framework byl zvolen pro urychlení vývoje a také z důvodu předchozích zkušeností s tímto frameworkem.

ThreeJS je knihovna pro 3D (i 2D) [22] vykreslování využívající *WebGL* [23] a je napsána v TypeScriptu. Obsahuje již zjednodušené API pro 3D vykreslování, např. obsahuje předpřipravenou kameru s možností rozhlížení se po scéně nebo disponuje připravenou třídou pro načítání objektů typu .obj. Tato knihovna byla zvolena kvůli rychlejšímu vývoji díky jednoduššímu API.

Kapitola 5

Zaznamenání běhu ray traceru

V předchozích kapitolách již bylo vysvětleno, za jakým účelem je takové zaznamenávání potřebné. V této kapitole bude uvedeno, jak bylo zaznamenávání běhu ray traceru implementováno.

Logování samotné není nijak složité, největším problémem je ale obrovský počet paprsků a tím pádem i velikost souborů, neboť s tím přichází problémy, jak tyto soubory načíst v prohlížeči, protože webové prohlížeče jsou omezeny pamětí, kterou mohou využívat. Příklad: renderuje se scéna ve 4K (3840×2160), tedy s minimem 8 294 400 paprsků (pro každý pixel jeden paprsek). V rámci možnosti načítání souborů v prohlížeči je dobré paprsky rozdělit do více souborů, ideálně aby každý soubor obsahoval stejný počet zásahů a nepřesahoval velikost 10 MB. Za předpokladu, že každý zásah obsahuje pouze základní informace a každý první zásah v paprsku bude obsahovat v uživatelem definovaných hodnotách (`custom_properties`) pozici pixelu paprsku `pixel_x` a `pixel_y`, kdy je velikost zásahu 400 B, bude limit počtu zásahů pro jeden soubor 25 000. Další nevýhodou obrovského souboru je samotné načítání v prohlížeči, musel by se načítat po částech, což by nebyl takový problém, ale musel by být použit nějaký jiný JSON parser, který by toleroval chyby. Soubor by totiž obsahoval jedno veliké pole a pokud by byl načítán po částech, tak žádná část nebude zcela validní. U takové části bude buď chybět začátek pole, konec pole a nebo obojí, a proto běžný JSON parser vyhodí chybu, jelikož to není validní JSON. Zásah paprsku v JSON souboru může vypadat následovně.

```
{
   "custom_properties": {
      "ior_env": 1.000,
      "type": "E"
   },
   "depth": 1,
   "direction_vector_x": -0.943, "direction_vector_y": -0.170, "direction_vector_z
       ": 0.286,
   "distance_t": 2.840,
```

```
   "normal_vector_x": 0.008, "normal_vector_y": -0.010, "normal_vector_z": 0.016,
   "origin_x": 3, "origin_y": 0, "origin_z": 0
}
```

Listing 5.1: Zásah paprsku ve formátu JSON. Na této ukázce lze vidět, že se pro zásah ukládají nejdůležitější informace a to hloubka, směr, vzdálenost, normála, počáteční bod a vlastnosti specifikované uživatelem

Tento RTLogger lze považovat za knihovnu samotnou. Vše je obaleno v namespace rtlogger, který kromě třídy RTLogger obsahuje také struktury inspirované z knihovny Intel embree. Tyto struktury vypadají následovně.

```
struct Ray
{
   float origin_x = 0;
   float origin_y = 0;
   float origin_z = 0;

   float direction_vector_x = 0;
   float direction_vector_y = 0;
   float direction_vector_z = 0;

   float distance_t = 0;
};

struct Hit
{
   // normal, material properties
   float normal_vector_x = 0;
   float normal_vector_y = 0;
   float normal_vector_z = 0;

   nlohmann::json material_properties;
};

struct RayHit
{
   Ray ray;
   Hit hit;
   int depth;
```

```
};
```

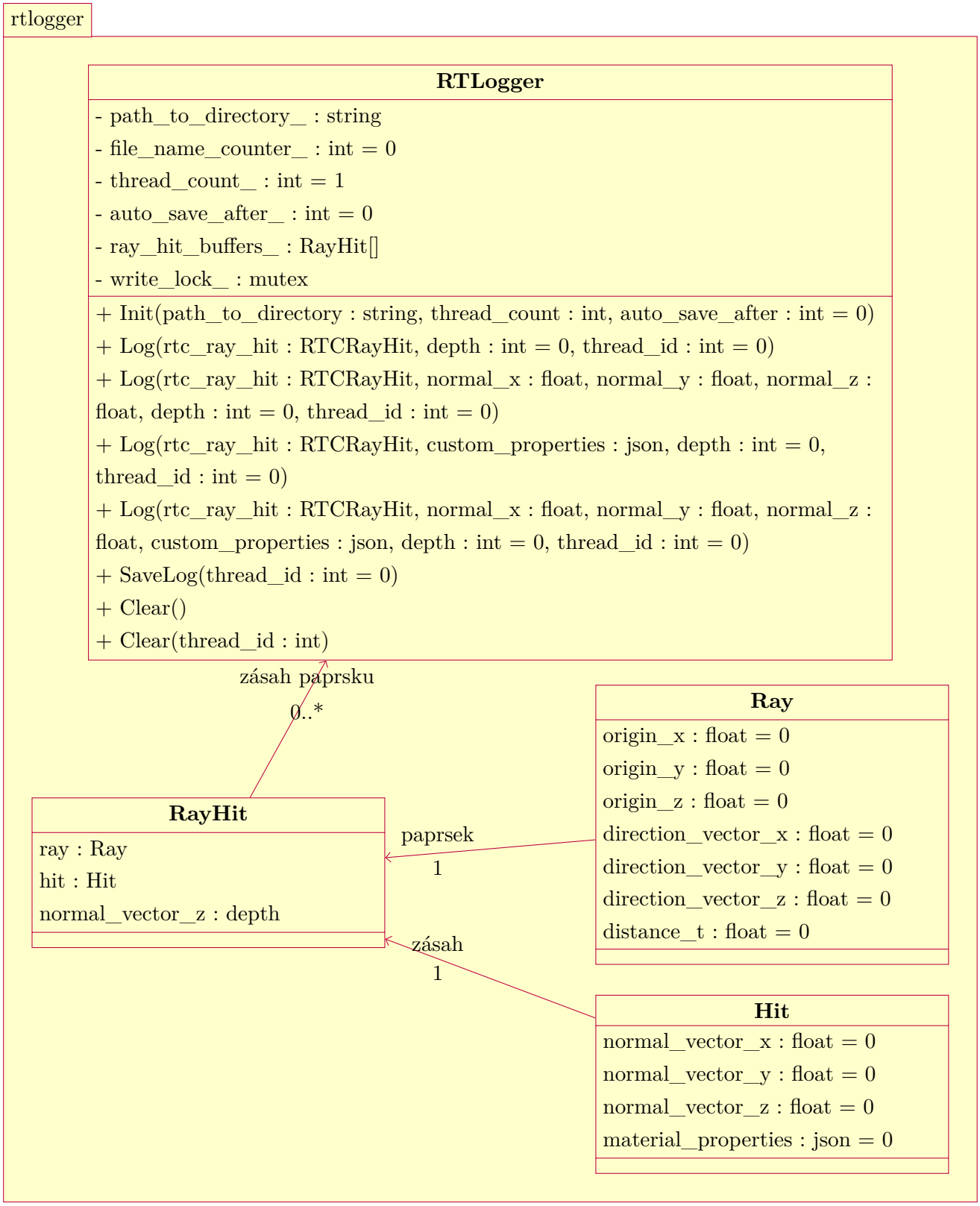
Listing 5.2: Struktury paprsků v Loggeru. V tomto kódu je ukázáno, že struktury obsahují všechny informace, které se posléze převádějí do formátu JSON

Jsou zde tři základní struktury a to Ray, Hit a RayHit.

- Ray - obsahuje informace o samotném paprsku: odkud vychází, jeho směr a vzdálenost dopadu
- Hit - obsahuje informace o dopadu paprsku, konkrétně normálu od místa dopadu, zde se přidal atribut `custom_properties`, který obsahuje uživatelem definovaný JSON o dalších informacích s povinnou informací jako je typ paprsku, dále může obsahovat například typ materiálu, pozici pixelu, pro který se tento paprsek počítal apod.
- RayHit - tato jednoduchá struktura spojuje předchozí dvě struktury a přidává atribut 'depth', což značí pořadí paprsku, např. při jednom paprsku, který se třikrát odrazil, bude mít první RayHit hloubku 1 a poslední 3

Třída RTLogger obsahuje čtyři základní metody a to Init, Log, SaveLog a Clear. V metodě Init se specifikuje cesta ke složce, do které se budou ukládat logovací soubory, počet vláken, které budou zapisovat pomocí RTLoggeru a nakonec počet zásahů paprsků v bufferu, po kterých se automaticky uloží buffer do souboru. Tato metoda automaticky vytvoří cestu ke složce (pokud neexistuje) a také předpřiraví stejný počet bufferů jako vláken. Přetížená metoda Log poté slouží k samotnému logování, přijímá parametry jako RTCRayHit, JSON s různými hodnotami, hloubku zásahu a nakonec id vlákna. Tato metoda zkontroluje, jestli je velikost bufferu větší než limit, po kterém se má automaticky ukládat do souboru, a zároveň je aktuální paprsek v bufferu celý (což se pozná podle hodnoty hloubky v paprsku, pokud bude tato hodnota rovná jedné, tak buffer aktuálně obsahuje všechny paprsky v celku), tak se zavolá metoda SaveLog a poté si předané parametry uloží do bufferu. Funkce SaveLog vyžaduje pouze jeden parametr a to id vlákna, podle kterého vybere buffer, který má uložit. Poté metoda převede položky v bufferu na JSON pole, které následně zapíše do souboru a buffer vyprázdní. Soubory jsou pojmenovány stylem `ray_<id_vlákna>_<čítač_souborů>`. Díky tomuto zápisu se zachová posloupnost paprsků pro každé vlákno, což lze využít pro budoucí implementaci možnosti výběru úseku/sekce pro vykreslení a vyhledávání paprsků. Nakonec RTLogger disponuje přetíženou metodu Clear, která vyprázdní buďto všechny buffery, nebo pouze buffer pro konkrétní vlákno, pokud je metodě předán parametr s id vlákna.

Na následujícím třídním diagramu lze vidět, že třída RTLogger v metodě Log přijímá parametr typu RTCRayHit, ale typ bufferu používá `RayHit[]` z toho důvodu, že struktury RTCRay, RTCHit a RTCRayHit patří pod knihovnu Intel Embree. Struktury z této knihovny ale neobsahují žádnou proměnnou, ve které by bylo možno ukládat JSON s vlastními hodnotami. Z tohoto důvodu jsou tyto struktury v metodě Log převáděny na struktury Ray, Hit a RayHit.


rtlogger
RTLogger
- path_to_directory_ : string
- file_name_counter_ : int = 0
- thread_count_ : int = 1
- auto_save_after_ : int = 0
- ray_hit_buffers_ : RayHit[]
- write_lock_ : mutex
+ Init(path_to_directory : string, thread_count : int, auto_save_after : int = 0)
+ Log(rtc_ray_hit : RTCRayHit, depth : int = 0, thread_id : int = 0)
+ Log(rtc_ray_hit : RTCRayHit, normal_x : float, normal_y : float, normal_z : float, depth : int = 0, thread_id : int = 0)
+ Log(rtc_ray_hit : RTCRayHit, custom_properties : json, depth : int = 0, thread_id : int = 0)
+ Log(rtc_ray_hit : RTCRayHit, normal_x : float, normal_y : float, normal_z : float, custom_properties : json, depth : int = 0, thread_id : int = 0)
+ SaveLog(thread_id : int = 0)
+ Clear()
+ Clear(thread_id : int)
zásah paprsku
0..*
RayHit
ray : Ray
hit : Hit
normal_vector_z : depth
paprsek
1
zásah
1
Ray
origin_x : float = 0
origin_y : float = 0
origin_z : float = 0
direction_vector_x : float = 0
direction_vector_y : float = 0
direction_vector_z : float = 0
distance_t : float = 0
Hit
normal_vector_x : float = 0
normal_vector_y : float = 0
normal_vector_z : float = 0
material_properties : json = 0

Kapitola 6

Uživatelské rozhraní

V předchozích kapitolách byl nachystán návrh projektu a také RTLogger, který nám zpracuje výstup ray traceru do JSON souborů. V této kapitole bude představeno uživatelské prostředí, které programátorovi umožní tyto data zpracovávat a debugovat tím ray tracer.

6.1 Rozložení stránky

Stejně jako každá webová aplikace, se i tento webový ray tracing vizualizér skládá ze čtyř základních elementů a to hlavičky, menu, obsahu a patičky. Hlavička se jako každá jiná nachází úplně nahoře a aktuálně drží pouze informaci o názvu aplikace, tedy RayTracingVisualiser. Pod hlavičkou se na levé straně nachází menu, které zatím obsahuje pouze tři položky a to domovskou stránku Home, Visualiser (pro nás nejdůležitější) a About. Po pravé straně menu se nachází obsah a pod ním nakonec patička.

Na stránce Visualiser se v horní části obsahu nachází panel nástrojů 6.1 obsahující různé možnosti pro práci s paprsky nebo scénou. Je zde možnost načíst soubory jak paprsků, tak i .obj soubory obsahující informace o objektech. Dále je zde pole pro vyhledávání paprsků. Za vyhledáváním následuje tlačítko pro zamknutí myši, které následně schová myš a zamkne myš na jednom místě pro jednodušší pohyb ve scéně (odemknout lze stiskem klávesy ESC). Po zamknutí myši můžeme vidět tlačítko zastavit nebo spustit vykreslování, které se aktuálně používá pro debugovací účely. Další tlačítko smazat všechny vykreslené paprsky umožňuje vyčistit scénu od vykreslených paprsků. Následuje tlačítko schovat nebo zobrazit osy, tyto osy přestože zlepšují orientaci ve scéně, mohou občas překážet, a proto je zde toto tlačítko. Poté je zde tlačítko pro zastavení rotace kamery kolem bodu zásahu, za kterým se nachází tlačítko vyresetování pozice kamery pro případy, kdy scéna neobsahuje moc objektů a je snadnější se v ní ztratit. Na konci řádku jsou vypsány informace jako je počet snímků za vteřinu a pozice kamery. Druhý řádek obsahuje tlačítko pro výběr sekce k vyhledávání a za ním je číselný vstup pro nastavení počtu vláken, které se mají při vyhledávání použít.

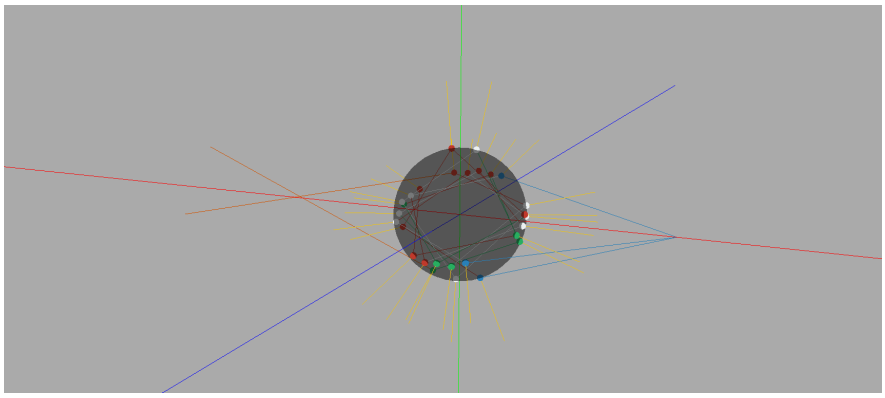
Obrázek 6.1: Panel nástrojů

Pod tímto panelem se nachází scéna 6.2 obsahující objekty a paprsky. Ve scéně se lze pohybovat pomocí kláves WASD pro pohyb vpřed, vlevo, vzad a vpravo a pomocí kláves Space a C se lze pohybovat nahoru a dolů. Každý paprsek je zobrazen čárou a koulí představující místo dopadu paprsku, přičemž pro lepší přehlednost jsou paprsky i barevně rozděleny dle jejich typu.

```
const HIT_TYPE = {
   EYE: {
      NAME: 'E',
      COLOR: '#2980B9', // modrá
   },
   DIFFUSE: {
      NAME: 'D',
      COLOR: '#27AE60', // zelená
   },
   SPECULAR: {
      NAME: 'S',
      COLOR: '#C0392B', // tmavě červená
   },
   LIGHT: {
      NAME: 'L',
      COLOR: '#ECF0F1', // světle šedá
   },
   INFINITE: {
      NAME: 'I',
      COLOR: '#D35400', // oranžová
   },
   NORMAL: {
      NAME: 'N',
      COLOR: '#F1C40F', // žlutá
   },
   UNKNOWN: {
      NAME: 'U',
      COLOR: '#7000E0', // fialová
   },
```

```
};
```

Listing 6.1: Typy paprsků a jejich barvy. Zde je zobrazeno, že pro základní typy paprsků jsou předdefinovány různé barvy pro rozlišení zásahů paprsků ve scéně



Obrázek 6.2: Scéna. Na obrázku je vykreslena scéna obsahující jednu kouli, osy x, y, a z a tři paprsky o deseti zásazích

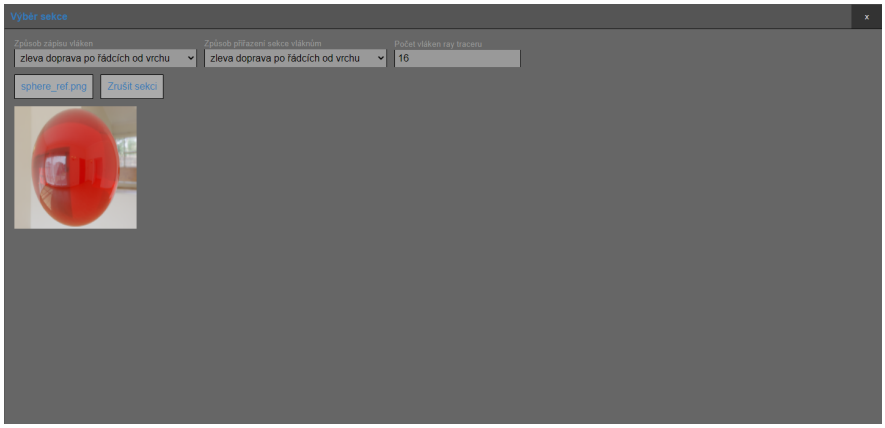
Po kliknutí na kouli se zobrazí okno s informacemi o konkrétním paprsku 6.3, toto okno lze zavřít nebo minimalizovat. Kromě informací o zásahu také obsahuje tlačítka pro otáčení kamery kolem zásahu a počátku, také tlačítko pro schování a zobrazení koule reprezentující dopad paprsku a nakonec tlačítko pro zobrazení okna s informacemi o předchozím a následujícím zásahu.

Obrázek 6.3: Okno s informacemi o zásahu paprsku. Na obrázku výše lze vidět okno s informacemi o zásahu paprsku, kde se na levé straně nachází základní informace a pod nimi se nachází uživatelem specifikované hodnoty. Na pravé straně se nachází tlačítka s možnostmi zobrazit předchozí/následující zásah, otáčet kameru kolem počátku/zásahu a tlačítko pro zobrazení/schování koule reprezentující zásah paprsku

6.2 Výběr sekce

V této části bude popsána implementace výběru sekce pro vyhledávání. Tato metoda pracuje s předpokladem, že osa y je oproti standartu zrcadlově převrácená, to znamená, že pro Full HD rozlišení se pixel na pozici `x: 0, y: 0` bude nacházet v levém horním rohu obrazovky a pixel s pozicí `x: 1919, y: 1079` se bude nacházet v pravém dolním rohu obrazovky. `RTLogger` je na tuto metodu připraven ukládáním pořadí souboru a id vlákna do názvu souboru.

V panelu nástrojů se nachází tlačítko pro výběr sekce pro vyhledávání a po kliknutí na toto tlačítko se zobrazí dialog 6.4, ve kterém uživatel zadá, jakým způsobem byly paprsky zpracovávány vlákny. Na výběr jsou možnosti jako zleva doprava po řádcích svrchu, zprava doleva po řádcích svrchu, zleva doprava po řádcích zdola, zprava doleva po řádcích zdola, zleva doprava po sloupcích svrchu, zprava doleva po sloupcích svrchu, zleva doprava po sloupcích zdola, zprava doleva po sloupcích zdola. Tato informace je využita při výběru vlastní sekce, kde se podle typu zpracování paprsků vlákny vyberou soubory, které obsahují paprsky nacházející se v uživatelem vybrané sekci. Obdobným způsobem je od uživatele požadováno, aby specifikoval jak byla vláknům přiřazena sekce, tato informace je důležitá, protože díky ní lze zjistit, které soubory se budou muset prohledávat, například při nastavení zleva doprava po řádcích víme, že soubory s prvním vláknem obsahují paprsky v levé horní obrazovce. Dále je zde možnost pro specifikace počtu použitých vláken, pokud ovšem uživatel již předtím vybral soubory s paprsky, tento počet vláken bude automaticky vyčten z názvu souborů. Poté se zde nachází tlačítko pro nahrání referenčního obrázku, který se posléze pod tímto tlačítkem zobrazí. Za tímto tlačítkem se nachází tlačítko pro zrušení výběru sekce.



Obrázek 6.4: Výběr sekce. Na tomto obrázku se nachází dialog pro výběr sekce. V horní části se nachází hlavička s názvem dialogu a pod ní se nachází obsah dialogu

Po kliknutí na referenční obrázek se tento obrázek zobrazí v původní velikosti spolu s hlavičkou se dvěmi tlačítky a informacemi jako je velikost vybrané sekce a zda bude použita stromová struktura pro vyhledávání. Tlačítka slouží pro přepnutí mezi výběrem sekce podle přiřazení sekce vláknům 6.5 a vlastním výběrem sekce 6.6. Pro možnost specifikace vlastní sekce musí uživatel v paprscích

nechat zapsat do `custom_properties` pozici pixelu (stačí pouze pro první zásah v paprsku) jako `pixel_x` a `pixel_y`.

Obrázek 6.5: Výběr sekce pro vlákno. Na tomto obrázku jsou sekce rozděleny pro 16 vláken

Obrázek 6.6: Výběr vlastní sekce

Postup pro rozdělení obrázku do sekcí si ukážeme na příkladu, kdy je celkový počet vláken osm. Než začneme, musíme si uvědomit, že hledáme, jaký je počet sekcí na ose x a na ose y. Z toho vyplývá, že nejprve se získá odmocnina z celkového počtu vláken. Ta může vyjít jako celé číslo (například pro celkový počet vláken čtyři) a postup je ukončen, ovšem v našem příkladu je odmocnina z osmi přibližně 2,83. Odmocnina se zaokrouhlí na desítky nahoru a dolů a pokud součin těchto dvou výsledků je roven celkovému počtu vláken, postup se ukončí (například pro celkový počet vláken šest $celkovyPocetVlaken = 6; pocetX = 3; pocetY = 2; pocetX * pocetY = 6 = celkovyPocetVlaken;$) ale pro naše číslo osm to nevychází $celkovyPocetVlaken = 8; pocetX = 3; pocetY = 2; pocetX *$

$pocetY = 6; 6! = celkovyPocetVlaken;$. V našem případě se tedy pokračuje kontrolou, zda je celkový počet vláken prvočíslo, a pokud ano, rozdělí se sekce do sloupců dle celkového počtu (například pro číslo sedm). V našem případě toto neplatí a pokračuje se zjištěním dvou čísel, jejichž součin se bude rovnat celkovému počtu vláken a absolutní hodnota jejich rozdílu bude co nejmenší. Takové dvě čísla lze použít pro naznačení sekcí, jelikož větší číslo se použije na osu x a menší (nebo stejné) číslo na osu y. Přesto ale toto řešení nemusí zajistit podporu pro všechny možnosti, pro naše potřeby je ale dostačující.

Jestli je číslo prvočíslo zjistíme pomocí jednoduchého algoritmu, který projde všechna čísla od dvou po a včetně polovinu čísla, které kontrolujeme. Pro každé číslo z iterace zjistíme, jestli kontrolované číslo podělené číslem z iterace vyjde beze zbytku a pokud ano, kontrolované číslo není prvočíslem a algoritmus je ukončen.

Obdobným způsobem se získávají i dvě celá čísla, jejichž součin je roven specifikovanému číslu a zároveň je absolutní hodnota jejich rozdílu nejmenší. Opět se iteruje od dvou po polovinu kontrolovaného čísla a kontroluje se zbytek po dělení. Zde přichází rozdíl v algoritmu, kdy při zjištění, že zbytek po dělení je nula, se získá výsledek dělení kontrolovaného čísla a čísla z iterace a rozdíl těchto dvou čísel. Následně se zkontroluje, jestli je tento rozdíl menší než čísla uchovaná z jiné iterace a nebo tato čísla nejsou nastavená. Tento algoritmus má v našem projektu smysl pouze pro kontrolu čísla pět nebo více.

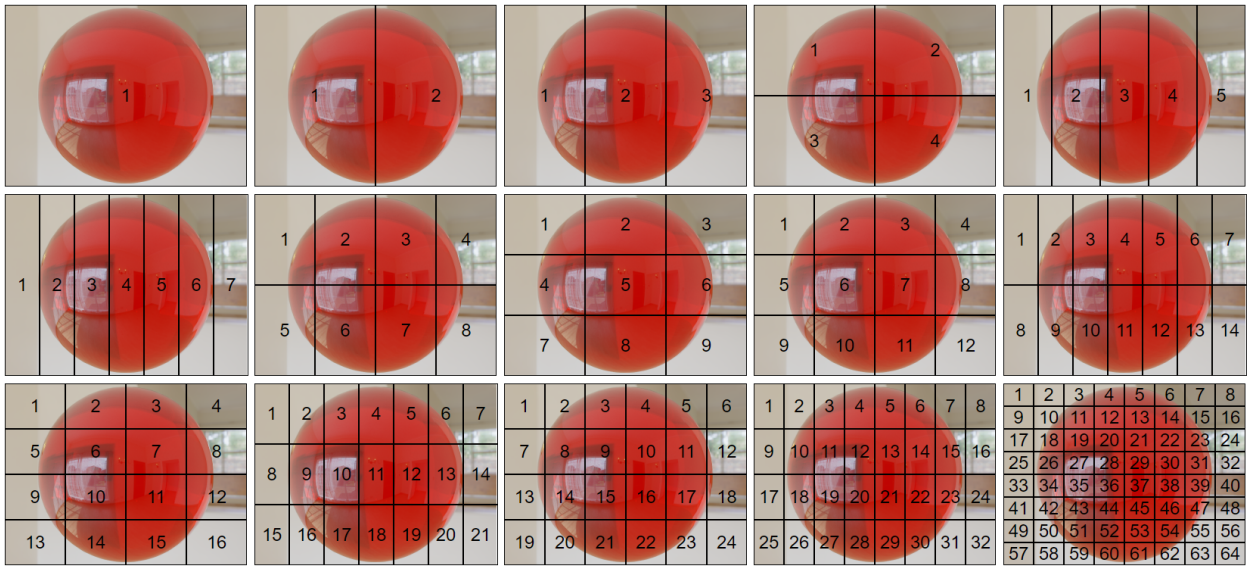
Potom, co jsou již zjištěny sekce pro vlákna, se uživateli umožní obdélníkovým výběrem vybrat konkrétní sekci. Pro použití stromové struktury k vyhledávání bude přednastaven limit na 10 000 pixelů, aby bylo zabráněno spadnutí prohlížeče kvůli nedostatku paměti. Sekce může být vybrána i větší než je tento limit, ale poté bude uživateli oznámeno zrušení použití stromové struktury pro vyhledávání. Po výběru konkrétní sekce bude uživatel moci buďto kliknout na tlačítko zrušit sekci, nebo dialog zavřít. Pokud dialog pouze zavře a vybraná sekce nepřesáhne nastavený limit pro použití stromové struktury, bude po prvním vyhledání sekce načtena do stromové struktury a tato struktura bude používána k vyhledávání, dokud uživatel nevybere jinou sekci nebo výběr sekce nezruší.

6.3 Vue

Zde si popíšeme části Vue jako jsou mixiny, direktivy, pluginy a komponenty. Také si popíšeme jak byly implementovány a jakou funkci plní v uživatelském rozhraní.

Direktivy se zapisují jako atributy v html pouze s prefixem `v-` [18] a slouží k úpravám vlastností elementu, kterému jsou přiřazeny. Pro životní cyklus direktiv jsou připravené funkce `bind`, `inserted`, `update`, `componentUpdated` a `unbind`.

Movable zajišťuje možnost pohybu elementu, na kterém je nasazen. Tato direktiva svému elementu nasazuje posluchače na události jako jsou pohyb myši, držení a puštění tlačítka myši.



Obrázek 6.7: Sekce pro vlákna. Na tomto obrázku jsou zobrazeny některé možnosti rozdělení sekcí pro různý počet vláken

Pluginy umožňují přidat vlastnosti přímo do prototypu Vue objektu [21], toto se provede tak, že v importovaném souboru se exportuje výchozí funkce, která přijme parameter Vue, do jehož prototypu lze posléze přidat potřebné funkce, třídy nebo objekty, které poté budou dostupné globálně ve všech komponentách a mixinech.

Collision poskytuje dvě funkce a to `detectRectangleWindowCollision` a `handleRectangleWindowCollision`. První funkce přijímá dva parametry `rect` a `box`, oba parametry jsou objekty obsahující čtyři hodnoty a to pozici `x`, `y` a velikost `width`, `height`. Funkce kontroluje, jestli a z jakých stran `boxu` `rect` koliduje. Zde by mohl nastat problém, pokud by byl rect větší než box, ale v našem použití se box považuje za okno prohlížeče a nepředpokládá se, že by se tento projekt používal na rozlišení menší než Full HD ($1920x1080$). Pokud tato funkce najde kolizi, vrátí funkce objekt obsahující názvy stran anglicky a k nim hodnotu, jestli je na této straně kolize. Objekt může vypadat takto: `{ left: true, right: false, top: true, bottom: false }`. Druhá funkce přijímá tři parametry `rect`, `collision` a `box`. Parametry `rect` a `box` jsou stejné jako u funkce první a parametr `collision` je výsledek předchozí funkce.

RayHit poskytuje třídu `RayHit` a funkci `getHitTypeKey`. Tento plugin také obsahuje objekt `HIT_TYPE` 6.1. Třída `RayHit` uchovává informace o zásahu paprsku a umožňuje takový zásah vykreslit spolu s koulí v bodě zásahu. Funkce `getHitTypeKey` přijímá jeden parametr `type`, který může obsahovat hodnoty typu zásahů jako jsou *EDSL*. Funkce poté najde příslušný klíč z objektu `HIT_TYPE` podle jména typu.

RayTree poskytuje třídu `RTNode` a funkci `getTypeMap`. Třída `RTNode` 7.1 uchovává informace o svém typu, rodiči, datech (paprscích) a také obsahuje své potomky. Tato třída umožňuje vybrat všechny potomky podle typu `E D S L ?`. Funkce `getTypeMap` přijímá parametr `ray` obsahující pole zásahů pro daný paprsek. Poté tato funkce vybere postupně všechny typy zásahů, první (typ `E`) odebere a vrátí nové pole s typy zásahů.

Utils disponuje funkcí `generateUID`, která přijímá jeden parametr `length` definující délku náhodného textového řetězce, který má být vygenerován a následně vrácen.

Komponenty jsou jádrem Vue a lze je rozdělit na tři části: HTML, CSS a JS [19]. V první části (HTML) jsou popsány elementy obalené elementem `<template>`. CSS se píší stejně jako v jiných HTML souborech a to pomocí tagu `<style>`, nebo přímo jako atribut elementu. Na konci je JS část, která je zabalená v tagu `<script>`. V této části se exportuje výchozí objekt, který může obsahovat následující z nejdůležitějších klíčů:

- mixins - pole obsahující všechny mixiny, které se mají pro tuto komponentu použít
- props - objekt, ve kterém se specifikují atributy komponenty a jejich vlastnosti jako je typ, výchozí hodnota nebo jestli jsou povinné
- data - funkce vracející objekt klíčů a hodnot, které jsou posléze dostupné jak v HTML tak i ve výchozím exportovaném objektu
- computed - objekt obsahující funkce nebo objekty, název funkcí nebo objektů lze použít v HTML nebo ve zbytku JS komponenty jako proměnnou, objekty mohou obsahovat dvě funkce a to `get` a `set` (tato funkce přijmá jeden parametr, kterým je hodnota přiřazená názvu tohoto objektu), pokud je specifikována pouze funkce bez objektu, je považována za funkci `get`
- created - funkce z životního cyklu komponenty, která je zavolána po vytvoření komponenty, je vhodná pro inicializaci komponenty a HTTP požadavky
- mounted - funkce z životního cyklu komponenty, která je zavolána po přidání komponenty do DOM objektu
- methods - objekt obsahující funkce, které jsou dostupné jak v HTML, tak i v ostatních funkcích komponenty včetně `created` a `mounted`

Mezi další funkce z životního cyklu komponenty patří `beforeCreate`, `beforeMount`, `beforeUpdate`, `updated`, `beforeDestroy`, `destroyed` a `activated`, ale ty pro naše řešení nejsou tolik důležité.

Dialog vytváří element překrývající všechny ostatní elementy. `Dialogu` lze pomocí *slotů* [24] předat obsah dialogu. Dialog sám o sobě disponuje hlavičkou obsahující titul vlevo a tlačítko zavřít vpravo.

InputFile obaluje element `input` typu `file` a zobrazuje, zda je vybrán nějaký soubor nebo soubory.

Window je komponenta využívající direktivu `Movable` a obdobně jako komponenta `Dialog` vytváří element překrývající všechny ostatní a je jí předáván obsah pomocí slotů. Taktéž obsahuje titul vlevo a tlačítka minimalizovat a zavřít vpravo.

RayInfo aplikuje komponentu `Window` a v něm obsahuje informace o zásahu paprsku, který je jí předán.

Toolbar zobrazuje již zmíněný panel nástrojů 6.1 zobrazující různé akce a vlastnosti.

Mixiny jsou to samé jako komponenty, pouze bez HTML části [20], jsou vhodné buďto pro rozdělení složité komponenty na menší části, nebo pro často se vyskytující se stejné části kódu, které lze posléze zabalit do mixinu a pouze jej v ostatních komponentách použít.

Controls využívá třídu `PointerLockControls` z knihovny ThreeJS a naslouchá událostem jako jsou stisk klávesy nebo tlačítka myši a pohyb myši. Na tyto události poté tento mixin reaguje a umožňuje tak pohybovat kamerou ve scéně.

RayLoader poskytuje funkce pro načítání souborů a následnému převodu do JS objektu, případně i načtení do stromové struktury `RTTree`.

RaySearch disponuje funkcemi pro analýzu vyhledávacího formátu, vyhledávání v souborech a porovnání zásahů paprsků.

6.4 Problémy a jejich řešení

Při vývoji uživatelského rozhraní byly nalezeny různé problémy, například problém s ovládáním, kdy původně pro pohyb dolů byla přiřazena klávesa CTRL a pro pohyb vpřed klávesa W, jenomže při pohybu vpřed a dolů naráz vznikne klávesová zkratka CTRL + W, která zavře aktuální záložku v prohlížeči. Z tohoto důvodu byla změněna klávesa pro pohyb dolů z CTRL na klávesu C. Další nepříjemnou vlastností bylo, že při stisku mezerníku, který byl použit pro pohyb nahoru, se stránka posunula dolů. Toto šlo jednoduše vyřešit pomocí zabránění výchozího chování prohlížeče při stisku mezerníku. Také byl nalezen problém, kde při psaní do vyhledávání kamera reagovala na stisknuté klávesy, například při psaní písmena D se kamera pohla doprava. Toto bylo nakonec vyřešěno ignorováním stisknutých kláves, pokud je v dokumentu aktivní nějaký element typu input. Mezi další nepříjemnosti lze zařadit možnost posunutí okna s informacemi o zásahu paprsku mimo obrazovku, což bylo nakonec vyřešeno detekcí kolize okna s informacemi s okrajem okna prohlížeče. V momentě,

kdy se pokusí uživatel přesunout okno s informacemi mimo okno prohlížeče, se okno s informacemi zastaví na okraji okna prohlížeče.

Kapitola 7

Vyhledávání

V této kapitole bude popsáno jakým způsobem funguje vyhledávání v paprscích, jak bylo implementováno urychlení vyhledávání a také obsahuje testy a měření rychlosti vyhledávání.

7.1 Formát

Aby bylo uživateli umožněno v paprscích nějakým způsobem vyhledávat, je potřeba si nadefinovat nějaký formát, podle kterého budeme vyhledávat. Vyhledávací formát má následující syntaxi: `<typ>[:[výchozí parametr][(pole_klíčů_a_hodnot)]][násobič]` a může vypadat například takto: `E D D:(vlastnost_x:'hodnota';vlasnot_y:0.5;vlastnost_z:[0.1,0.2,0.3]) ?*`. Tento formát bude zpracován pomocí regulárních výrazů.

Typ může být jeden z následujících:

- ? - jakýkoliv
- E - oko (eye)
- D - difúzní (diffuse)
- S - spekulární (specular)
- L - světlo (light)

Počet opakování může být jedno z následujících:

- * - nula nebo vícekrát
- + - jednou nebo vícekrát

7.2 Princip vyhledávání

Nejprve se začíná analýzou formátu pro vyhledávání, poté se podle nastaveného počtu vláken vytvoří *Web Workery* [25] a v každém vlákně se prochází všechny soubory. Jelikož je množství souborů s paprsky veliké už jenom pro full HD rozlišení (při použití odmocniny z velikosti rozlišení to pro full HD vyústí v 1 440 souborů), je načítání achillovou patou rychlosti vyhledávání. Před první iterací se proto vytváří fronta pro načítání souborů s jedním souborem a v každé iteraci se do fronty přidává další soubor. Obsah souboru se převede z JSON formátu na JS pole, které se následně prochází. Pro každý zásah se kontrolují vlastnosti z vyhledávacího formátu a pokud vlastnost zásahu tomuto formátu vyhovuje, pokračuje se dokud nedojdou vlastnosti z vyhledávacího formátu nebo nedojdou zásahy paprsků. Při procházení pole vlastností z vyhledávacího formátu se při každé shodě zvyšuje index zásahu o jedna. Po úspěšném dokončení se opět zvyšuje index zásahu, dokud se nenarazí na zásah, který již patří do jiného paprsku. V tento moment se zásahy paprsku přidají do pole pro vykreslení. V opačném případě se tyto zásahy pouze přeskočí.

Výhodou tohoto vyhledávání je možnost prohledání všech souborů. Nevýhodou je, že každý soubor se musí nejprve načíst a převést z formátu JSON do JS pole, což algoritmus velice zpomaluje.

7.3 Vyhledávání v sekci

Jelikož obyčejné vyhledávání je zdlouhavé a na snímku vykresleném ray tracerem lze někdy chybu spatřit díky zvláštním barvám nebo podivným odrazům, bylo pro urychlení navrženo vyhledávání v určité sekci. Jak již bylo v úvodu zmíněno, vlákna ray traceru mohou počítat paprsky v určitých sekcích (jak lze vidět na tomto obrázku 6.7) a přesně toho zde využijeme. Uživatel má možnost specifikovat, jakým způsobem byla vláknům přiřazena sekce a poté si může vybrat sekci, kde předpokládá, že se chyba nachází, a následně se v této sekci může vyhledávat.

7.4 Vyhledávání ve stromové struktuře

Pro rychlejší vyhledávání v takové sekci byla pro toto použití navrhnuta stromová struktura 7.1 pro uložení paprsků, do které se veškeré paprsky načtou a poté vyhledávání probíhá nad touto strukturou. Avšak sekce nemůže být příliš velká, pokud se má tato struktura použít, protože webové prohlížeče jsou omezeny pamětí. Kvůli tomuto stromová struktura nebude použita, pokud bude vybrána sekce s více než 10 000 pixely. Jak lze vidět v kódu 7.1, není to klasická stromová struktura, která by obsahovala pro jeden uzel pouze jednu hodnotu. Každý uzel v této struktuře uchovává pole paprsků s jejich zásahy odpovídající posloupnosti typu jejich zásahů. To znamená, že paprsky s posloupností zásahů `E D S L` se budou nacházet v hloubce tři této struktury. Za přepokladu, že stromová struktura bude rovnoměrně naplněna, bude každý uzel obsahovat 250 paprsků, z čehož každý paprsek bude obsahovat 10 zásahů a maximální hloubka posloupnosti zásahů bude tři, pak lze

tímto způsobem z vyhledávání pro tento konkrétní příklad eliminovat až 9 750 z 40 000 paprsků nebo 97 500 ze 100 000 paprsků. Vyhledávání se stává méně efektivním v momentě, kdy je ve vyhledávacím formátu použit jakýkoliv typ paprsku (`?`), protože poté se musí prohledat o dvě větve více a nebo v případě, kdy jsou použity násobiče.

7.4.1 Postup vyhledávání ve stromové struktuře

Podstatou vyhledávání ve stromové struktuře je vyfiltrovat pouze paprsky, které odpovídají posloupnosti typů zásahů z vyhledávacího formátu. Vyhledávání v stromové struktuře začíná zjištěním prvního typu z vyhledávacího formátu (typ oka je v tomto případě vynechán, protože každý paprsek začíná v oku) a získáním potomků aktuálního uzlu dle tohoto typu. Pokud je typ jakýkoliv, jsou vybráni potomci všech typů. V následujícím kroku se kontrolují násobiče, pokud je specifikován násobič jeden nebo více, tak se zkontroluje, jestli byli získáni nějací potomci nebo se jedná o poslední typ ze specifikované posloupnosti a potomci neobsahují data, pokud ano, vrátí se ve funkci prázdné pole. V opačném případě se vyberou všichni potomci (rekurzivně, tedy i potomci potomků atd.) dle specifikovaného typu, pokud se jedná o poslední typ z posloupnosti, vrátí se tyto uzly, jinak se pro každý uzel opakuje celá operace znovu. Část pro násobič nula nebo více se liší pouze začátkem, kde se při kontrole, jestli byli získáni nějací potomci dle typu, pokračuje kontrolou, jestli se nejedná o poslední typ z posloupnosti a pokud ne, tak se rekurzivně volá celá funkce s následujícím typem. V druhém případě se opět vrací prázdné pole. Nakonec, pokud není specifikován žádný násobič, tak se zkontroluje, jestli se nejedná o poslední typ z posloupnosti a pokud ne, tak se pro každého získaného potomka volá rekurzivně celá funkce a jejich výsledky se spojí do jednoho pole, které je následně vráceno. Jinak se vrátí pouze uzly, které obsahují data. Tímto získáme všechny uzly podle specifikované posloupnosti zásahů a může se v jejich paprscích následně vyhledávat stejným způsobem, jako bylo zmíněno výše.

7.4.1.1 Zpracování různých typů a násobičů

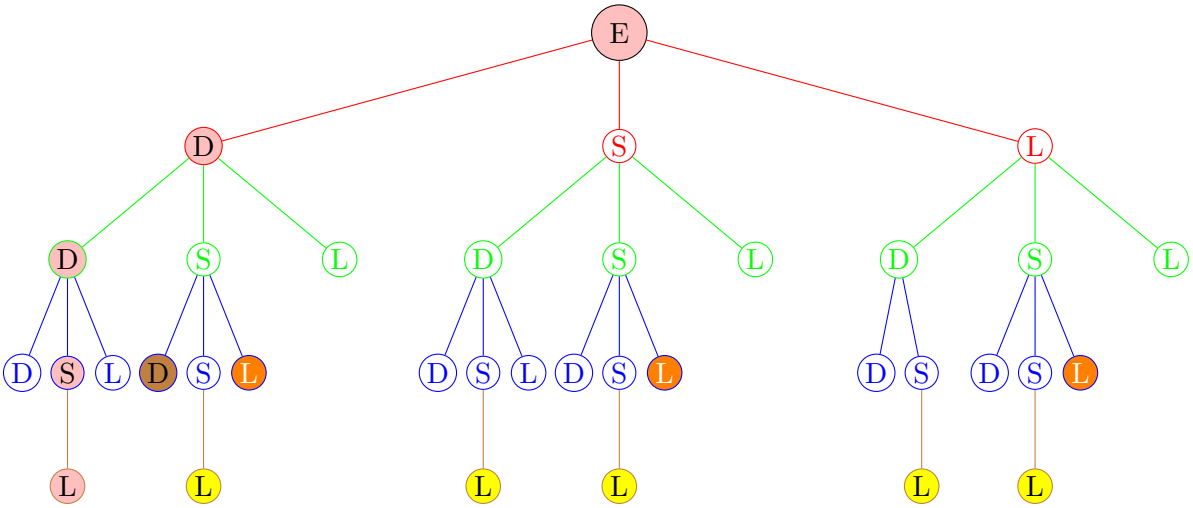
V následujícím grafu je znázorněno vyhledávání formátu `E ?+ S L`, konkrétně je zde vyobrazen proces výběru uzlů pro jakýkoliv typ. Uzly jsou barevně rozděleny podle čísla iterace, tedy červená barva představuje první iteraci, kdy se vybírají potomci jakéhokoliv typu, tedy v našem stromě jsou to D, S a L. Pro tyto uzly se operace opět opakuje a pokračuje se výběrem dalších všech typů (znázorněno zelenou, modrou a hnědou barvou). Tato část algoritmu by mohla být teoreticky vylepšena tím, že by se při načítání stromové struktury pro každý uzel uložila délka jeho nejdelší větve. To lze využít tak, že by se za každým násobičem + ve vyhledávacím formátu spočítal minimální počet následujících uzlů, které by byly potřeba pro splnění vyhledávacího formátu a v každém uzlu by se provedla kontrola, zda tento uzel obsahuje alespoň tento minimální počet. V opačném případě by se mohl výběr pro tento uzel zastavit. Minimální počet následujících uzlů by se dal získat součtem počtu vyhledávacích typů vynásobených jejich násobičem, kde by se násobič

$*$ bral jako nula a násobič $+$ jako jedna, pro tento příklad v první iteraci by bylo toto minimum tři $jakykolivTyp = 1; S = 1; L = 1; jednaNeboVice = 1; jakykolivTyp * jednaNeboVice + S + L = 3;$.

7.4.1.2 Zpracování konkrétních typů

V předchozí části bylo ukázáno, jak vybrat uzly v případě použití jakéhokoliv typu a násobiče. Nyní bude předvedeno, jak probíhá druhá část, kde je pouze konkrétní typ a žádný násobič. Všechny uzly, které se musí zkontrolovat, jsou barevně zobrazeny na předchozím grafu. Začneme první iterací, kdy byly získany zelené uzly D, S a L. Pro každý z těchto uzlů zkontrolujeme, zda obsahují potomky v pořadí S L a uzly, které budou přidány do finálního vyhledávání označíme oranžovou barvou. U všech typů lze ihned vidět, že takové potomky obsahují, a proto jsou poslední uzly typu L přidány do finálního vyhledávání. V další iteraci proces opakujeme a výsledné uzly označíme barvou žlutou. Ve třetí a ve čtvrté iteraci již žádné uzly vyhovující zadání nenacházíme a ve finálním vyhledávání končí dvanáct uzlů. Zde je možné vylepšit algoritmus kontrolou, zda daný uzel obsahuje minimální počet potomků.

Poněvadž už jsou vyfiltrovány všechny uzly, které mohou potencionálně obsahovat paprsky vyhovující vyhledávacímu formátu, algoritmus může projít všechny paprsky v těchto uzlech a vyfiltrovat je podle dodatečných vlastností, jak již bylo zmíněno výše. Pro tento konkrétní příklad se takové filtrování provádět nemusí, neboť žádné dodatečné parametry nejsou ve vyhledávacím formátu specifikovány. Růžovou barvou jsou vyplněny uzly pro jeden konkrétní paprsek, který vyhovuje vyhledávacímu formátu.



Obrázek 7.1: Příklad stromové struktury pro uložení paprsků. Na obrázku se nachází ukázka stromové struktury s barevně rozlišenou hloubkou uzlů. Každý uzel může obsahovat pole paprsků, které odpovídají posloupnosti předchozích typů uzlů a aktuálního uzlu. To znamená, že paprsky s posloupností typů E D S D se budou nacházet v uzlu s hnědým pozadím

Jak již bylo zmíněno, webové prohlížeče jsou omezeny pamětí a tudíž nelze vyhledávání v struktuře uplatnit pro všechny paprsky, protože prohlížeč před načtením všech paprsků spadne. Limit, pro jaký počet paprsků by tuto metodu šlo teoreticky použít, se ovšem může na různých zařízeních a v různých prohlížečích lišit.

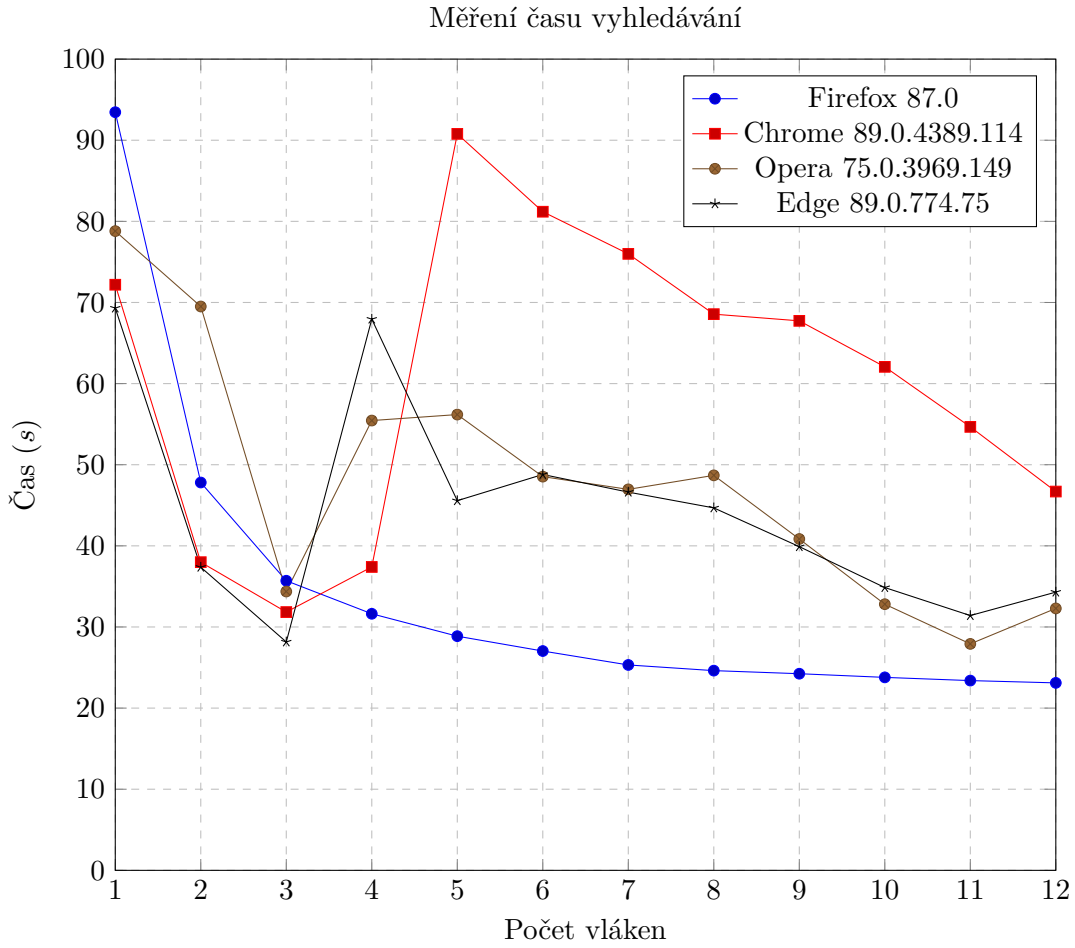
```
class RTNode {
   constructor(parent, type) {
      this.data = [];
      this.children = [];
      this.parent = parent;
      this.type = type;
   }
}
```

Listing 7.1: Stromová struktura pro uložení paprsků. V této ukázce je zobrazena třída RTNode, která obsahuje základní informace jako je typ uzlu, paprsky vyhovující posloupnosti typů uzlů a potomci

7.5 Testy a měření

Při testování rychlosti algoritmu byl pro vyhledávání ve všech souborech naměřen průměrný čas 249 ms pro 19 000 paprsků (24 MB) v jednom souboru, kde to ale silně zpomalilo načítání a převod formátu JSON do JS pole. Průměrná doba načtení souboru byla 231 ms, tedy samotné vyhledávání trvalo průměrně 18 ms. Při vyhledávání ve stejném počtu paprsků, ale rozdělených do více souborů (konkrétně 206 souborů), byl průměrný čas 495 ms z čehož 42 ms trvalo vyhledávání samotné a 453 ms trvalo načtení souborů. Zde lze jasně vidět výhoda při načítání pouze z jednoho souboru, což ale ve větším množství paprsků není bohužel možné. U většího množství paprsků odpovídajících výstupu ray traceru pro full HD snímek, byl počet souborů 1440 o celkové velikosti skoro 7 GB naměřen průměrný čas 71,66 s při použití pouze jednoho vlákna, z čehož 1,195 s trvalo vyhledávání a zbytek načítání souborů. Na tomto příkladu lze jasně vidět důvod, proč muselo být navrženo vyhledávání v sekci spolu se stromovou strukturou. Zajímavostí je, že při testování vyhledávání v development verzi se časy podstatně lišily, konkrétně u většího počtu paprsků byl naměřen průměrný čas 108,575 s z čehož vyhledávání trvalo 3,736 s. Taková odchylka byla zřejmě způsobena kvůli používání vue-development tools rozšíření pro webový prohlížeč. Následující graf zobrazuje, jak použití více jader zrychluje nebo zpomaluje vyhledávání v různých prohlížečích. Měření bylo prováděno na notebooku s šesti jádrovým procesorem Intel Core i7-8750H. Nejlépe je na tom prohlížeč Firefox s nejlepším časem 23,108 s při použití 12 vláken, ale zároveň je na tom nejhůře při použití jednoho vlákna s časem 93,456 s. Z tohoto důvodu je pro používání tohoto projektu doporučeno používat

webový prohlížeč Firefox při použití více než jednoho vlákna, a při použití pouze jednoho vlákna prohlížeč Edge. Pro ostatní prohlížeče lze doporučit používat pro vyhledávání tři vlákna.



Obrázek 7.2: Graf rychlosti vyhledávání. Na tomto grafu jsou zobrazeny rychlosti vyhledávání prohlížečů Firefox, Chrome, Opera a Edge pro různý počet vláken.

Kapitola 8

Závěr

V této poslední kapitole shrneme výsledky bakalářské práce a navrhneme vylepšení, která by se dala implementovat v budoucnu.

Cílem práce bylo umožnit uživateli zaznamenat cesty paprsků z ray traceru a poté nabídnout uživateli nahrát tyto záznamy do prohlížeče, kde uživateli budou k dispozici nástroje pro vyhledávání a následnou vizualizaci vykreslených paprsků.

Co se týče uživatelského rozhraní, bylo by dobré navrhnout lepší rozložení ovládacích prvků. Velmi by také pomohlo vytvořit ikonky pro jednotlivé tlačítka a následně tato tlačítka nahradit ikonkami, ke kterým by se pouze přidal popisek, který by se zobrazil při najetí myší na danou ikonku. Ve scéně by bylo možné přidat funkce sledování paprsku z pohledu první nebo třetí osoby a také by mohla být přidána možnost nastavení vlastního ovládání a přiřazení kláves. Do vyhledávacího formátu by bylo dobré přidat možnost specifikace větší, větší nebo rovno, menší a menší nebo rovno pro číselné hodnoty.

Pro maximální zrychlení vyhledávání a načítání souborů by možná stála za vyzkoušení knihovna *NodeGui* [26], která umožňuje vytvořit přímo aplikaci pro různé platformy, což by nám odemklo limit pro použití paměti RAM a tím pádem by se mohlo vyhledávání zrychlit uchováním paprsků přímo v paměti. NodeGui dokonce podporuje i framework Vue, takže přepsání kódu pro použití NodeGui by nemuselo být tak složité.

Projekt jako takový se nakonec povedl, protože uspěl v nejdůležitějších požadavcích a to zaznamenat cesty paprsků pomocí `RTLoggeru` a poté umožnit vyhledávání a vizualizaci těchto paprsků.

Literatura

1. ASHWORTH, Booen. *Ray Tracing* [online] [cit. 2020-07-03]. Dostupné z: https://www.wired.com/story/what-is-ray-tracing/.
2. RYER, A. D. *The Light Measurement Handbook*. International Light Inc., 1997.
3. *Hidden surface determination* [online] [cit. 2020-07-03]. Dostupné z: https://viscircle.de/guide-for-beginners-what-is-hidden-surface-determination/?lang=en.
4. *NVIDIA Nsight Graphics* [online] [cit. 2020-07-03]. Dostupné z: https://developer.nvidia.com/nsight-graphics.
5. *rtVTK README* [online] [cit. 2020-07-03]. Dostupné z: https://www.rtvtk.org/current/README.
6. *C++ OOP* [online] [cit. 2020-07-03]. Dostupné z: https://www.w3schools.com/cpp/cpp_oop.asp.
7. *C++* [online] [cit. 2020-07-03]. Dostupné z: https://www.cplusplus.com/.
8. *Intel embree* [online] [cit. 2020-07-03]. Dostupné z: https://www.embree.org/.
9. *prototype-based* [online] [cit. 2020-07-03]. Dostupné z: https://developer.mozilla.org/en-US/docs/Glossary/Prototype-based_programming.
10. *JavaScript* [online] [cit. 2020-07-03]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/JavaScript.
11. *First-class Function* [online] [cit. 2020-07-03]. Dostupné z: https://developer.mozilla.org/en-US/docs/Glossary/First-class_Function.
12. GAGLIARDI, Valentino. *event-driven* [online] [cit. 2020-07-03]. Dostupné z: https://www.valentinog.com/blog/event/.
13. *Node.js* [online] [cit. 2020-07-03]. Dostupné z: https://nodejs.org/en/.
14. *C++ JSON* [online] [cit. 2020-07-03]. Dostupné z: https://github.com/nlohmann/json.
15. *YAML* [online] [cit. 2020-07-03]. Dostupné z: https://yaml.org/.
16. *XML* [online] [cit. 2020-07-03]. Dostupné z: https://www.w3.org/XML/.

17. *Vue* [online] [cit. 2020-07-03]. Dostupné z: https://vuejs.org/.

18. *Custom Directives* [online] [cit. 2020-07-03]. Dostupné z: https://vuejs.org/v2/guide/custom-directive.html.

19. *Components Basics* [online] [cit. 2020-07-03]. Dostupné z: https://vuejs.org/v2/guide/components.html.

20. *Mixins* [online] [cit. 2020-07-03]. Dostupné z: https://vuejs.org/v2/guide/mixins.html.

21. *Plugins* [online] [cit. 2020-07-03]. Dostupné z: https://vuejs.org/v2/guide/plugins.html.

22. *ThreeJS* [online] [cit. 2020-07-03]. Dostupné z: https://threejs.org/.

23. *WebGL* [online] [cit. 2020-07-03]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API.

24. *Vue sloty* [online] [cit. 2020-07-03]. Dostupné z: https://vuejs.org/v2/guide/components-slots.html.

25. *Web Worker* [online] [cit. 2020-07-03]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers.

26. *NodeGui* [online] [cit. 2020-07-03]. Dostupné z: https://github.com/nodegui/nodegui.